

Compliance at the Point of Change

Unlocking developer efficiency in secure and compliant platforms



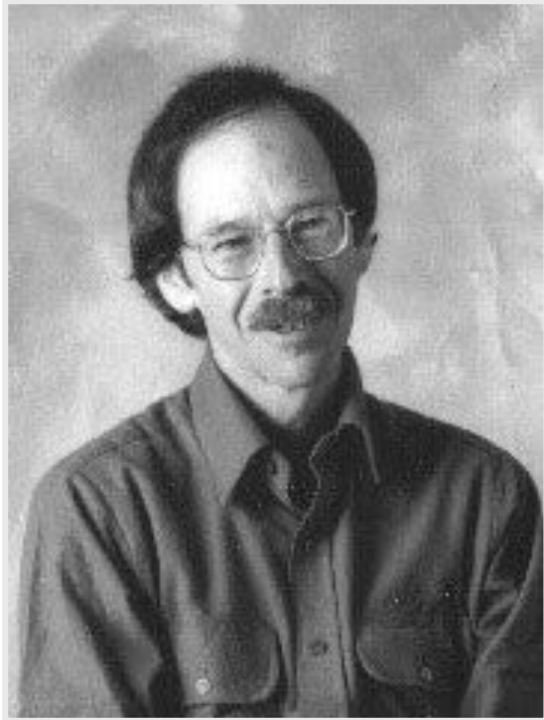
Preamble, a short story



TABLE 1

Skill Level Mental Function					
	NOVICE	COMPETENT	PROFICIENT	EXPERT	MASTER
Recollection	Non-situational	Situational	Situational	Situational	Situational
Recognition	Decomposed	Decomposed	Holistic	Holistic	Holistic
Decision	Analytical	Analytical	Analytical	Intuitive	Intuitive
Awareness	Monitoring	Monitoring	Monitoring	Monitoring	Absorbed

*A Five stage model of the mental activities involved in directed skill acquisition
Dreyfus & Dreyfus, P. 15*



Instructor pilots detect errors with greater accuracy than do student pilots, they are faster at detecting errors than students, and systematic cross-check patterns did not appear to be employed by instructor pilots while student pilots appeared to utilize systematic patterns.

This superior performance obtained despite the fact that the instructor pilots **did not use any detectable scanning pattern.**

Dreyfus and Dreyfus
The Scope, Limits, and Training Implications of Three Models
of Aircraft Pilot Emergency Response Behavior

“Use rules for novices, intuition for experts.”

- Andy Hunt, Pragmatic Thinking and Learning

“Having amazing developers who can produce high-quality code but having a process that does not enable them to work well will also not make projects succeed.”

- Sandro Mancuso, The Software Craftsman

DevSecOps & Compliance Enforcement



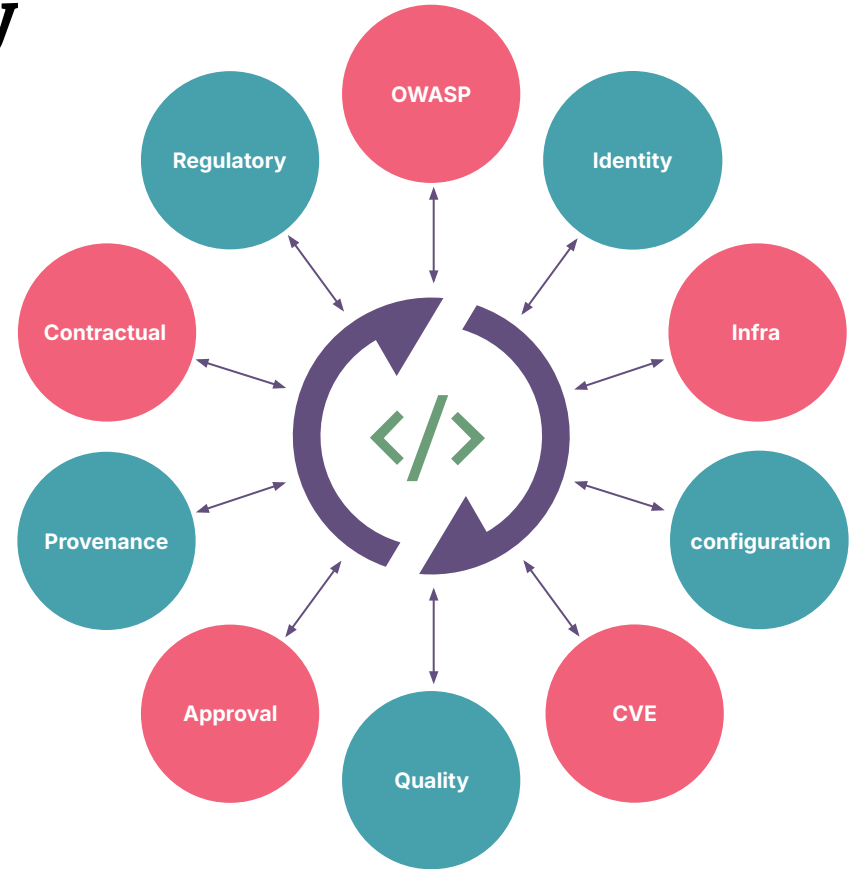
[In regulatory environments], average total cost of non-compliance is \$14.82 million, compared to a \$5.47 million cost of compliance



Ensuring a compliant delivery of software

There are many different stakeholders, regulatory factors, and technology standard that need to be met when releasing software.

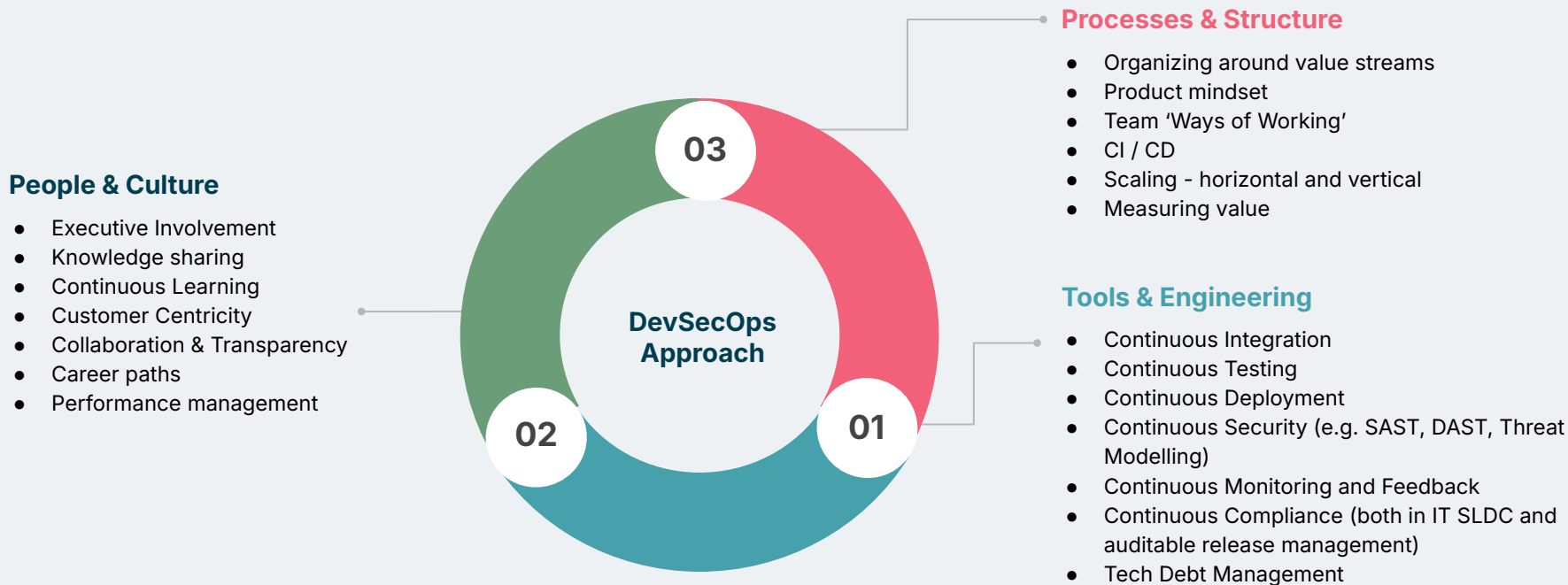
How to manage these different disciplines, contexts, and defining the requirements of a secure release is a constant struggle for most organizations.








DevSecOps

Platform Engineering and DevSecOps are evolving

As organisations move to adopt advanced IDP solutions, there is a clear opportunity to improve DevSecOps practices. However, in a DevSecOps model, people, culture and process are as important as tools and engineering



Key technical building blocks of a successful DevSecOps ecosystem

 Design	 Code	 Build	 Test	 Deploy / Monitor
Architecture review Manual Review	Code Review Application Security code review performed at the local setup using IDE plugins	Container Scans Scanning Docker Container images for known vulnerabilities	Application Security PenTesting Manual application testing to find security vulnerabilities	Container Image Scans Scanning Running Container images for known vulnerabilities
Threat Modeling Using ThreatModeler to auto generate possible threats	IDE Plugin for License Scan Scanning for license violations - OSC Review	License Violations Scan Scanning for license violations - OSC Review	Infrastructure Security Testing Testing for infrastructure specific vulnerabilities	Interactive Application Security Testing Runtime Protection for security issues
Training Business function specific security trainings based on tech stack	IDE Plugin - Code Scan Scanning for vulnerabilities in OSS libraries	Source Composition Scan Scanning for vulnerabilities in OSS libraries	Automation Testing Testing includes Regression Test, API Test , Contract Test , etc..	Configuration Review Identify security misconfigurations
	IDE Plugin - Runtime Scan Scanning for code issues - Runtime analysis	Artifact Management Managing of the build & binaries obtained from the pipeline processes		SIEM Monitor logs and application activities and alert
	Static Code Analysis A code analysis focused towards improvising code quality	Static Application Security Testing (SAST) Scanning for code issues		
	Pre-commit Secret Scan A Pre-commit scanning to prevent secrets getting committed	Dynamic Application Security Testing (DAST) Using fuzzing tools to check common defects at runtime		

DevSecOps maturity model

Capabilities levels	Security Inclusive Requirements	Threat Modelling	Static Code Scanning	Vulnerability Backlog Management	Dependency Management	Automated Testing
Novice	Security is an afterthought in feature development.	No threat modelling is ever performed.	No static code scanning exists.	No vulnerability backlog exists.	No dependency scanning exists.	All testing is performed manually and by a separate QA team.
Exploring	Functionality and security are considered separate requirements.	A stale/outdated threat model exists.	No process defined for improving static code scanning results.	No process defined for managing and prioritising the vulnerability backlog.	No process defined for remediating vulnerabilities identified in dependency scans.	Only functional automated tests exist.
Emerging	Selected features have security requirements specified with functional requirements.	Threat modelling performed infrequently independent of development.	Scanning results are acted upon when capacity permits.	Vulnerability backlog is prioritised on technical grounds separately from functional backlog.	Vulnerabilities identified in dependency scans remediated when capacity permits.	Automated security testing exists but separately from functional testing.
Growing	All features have security requirements specified with functional requirements.	Threat modelling of specific application/functionality conducted prior to development or maintenance.	Scanners integrated with CI/CD pipeline and break builds when certain thresholds are crossed.	A single prioritised iteration backlog is prepared with vulnerability fixes and features.	Dependency vulnerabilities are included in the Backlog and prioritised on the basis of business impact and associated risks.	Automated security testing is part of pre-release testing.
Leading	Security verification part of acceptance criteria of features.	Threat modelling part of SDLC.	Scanners integrated with IDEs to allow developers to fix issues prior to commit. Thresholds are progressively raised.	Business impact of vulnerabilities is determined and vulnerabilities prioritised on the basis of their business impact and associated risk.	Dev team actively practices minimalism of dependencies, removing unwanted dependencies and upgrading/patching the required dependencies to minimise business risks.	Security issues are resolved using TDD style approach by replicating vulnerabilities at unit and component test levels and then remediating them.

DevSecOps enables your policy as code engine

DevSecOps capabilities should be available in different ways and formats across different phases within the SDLC. The data that comes out of these tools is critical input into your policy-as-code or governance engine when it comes to security, quality, and vulnerability analysis.

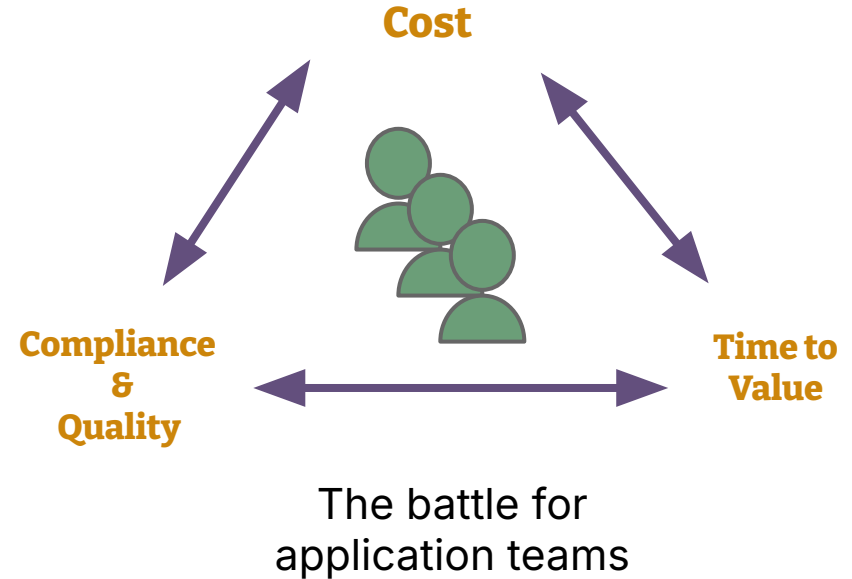
SBOM Artifact Store Storage of a comprehensive list of all SBOMs within your organization	Automated SAST Scanning for vulnerable libraries, secrets and license violations	Automated Code Quality Scanning A code analysis focused towards improving code quality	Artifact Signing and Verification Verifies artifacts being used are actually the ones you created
Automated SBOM Generation A comprehensive list of all software components within a container	Automated DAST Scanning for code issues - Runtime analysis	Automated IaC Scanning Scanning for security vulnerabilities or compliance violations	Signed commits Verifies developer committing code is actually that person
Software Composition Analysis Scanning for vulnerabilities in OSS libraries	Automated MAST Scanning for mobile app issues - Runtime analysis	Container Scanning (runtime) Scanning Running Container images for known vulnerabilities	Cloud Security Posture Management Detect, prevent and remediate cloud misconfigurations

Compliance & Governance

The pressure of the development triangle

Engineering leadership and the teams themselves are under constant pressure to deliver **high quality, compliant software** in the quickest and cheapest way possible.

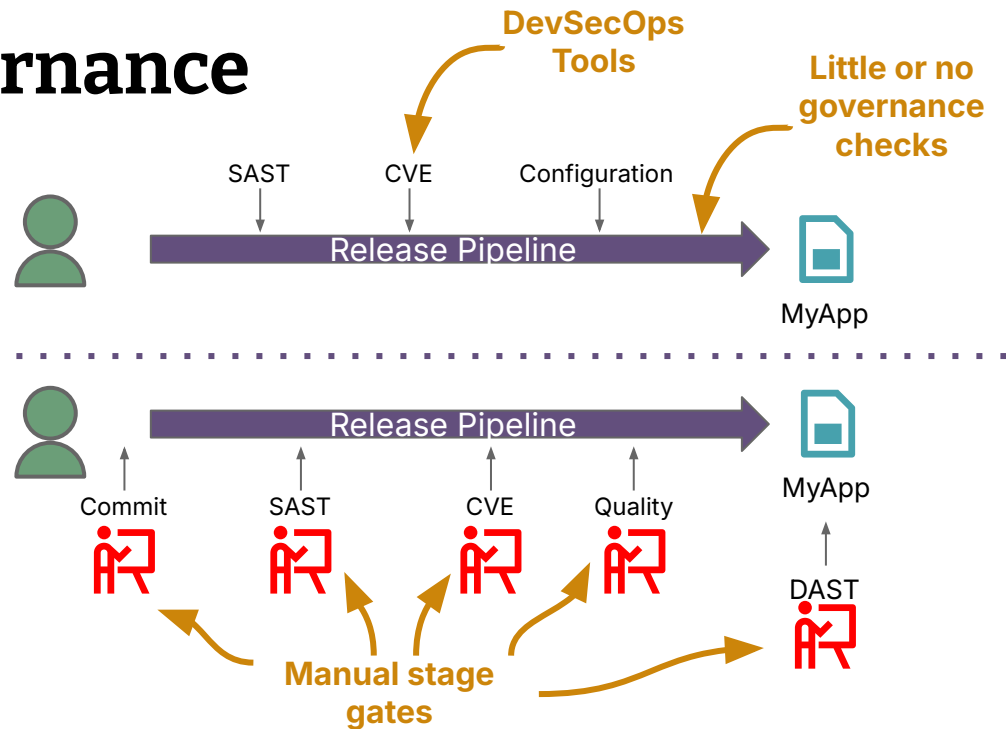
With any triangle **you only get to choose two sides**. Consciously or unconsciously, there are decisions being made that affect the final outcome with compliance and quality usually being left behind.



Common state of governance

These types of patterns are very common and come with a troublesome set of problems.

One has little to no governance while the other is very labor intensive and can take a long time to complete.



Problems occur late in the process and are owned by other teams

Checklist oriented, tool specific gates

Entire pipeline managed by another team

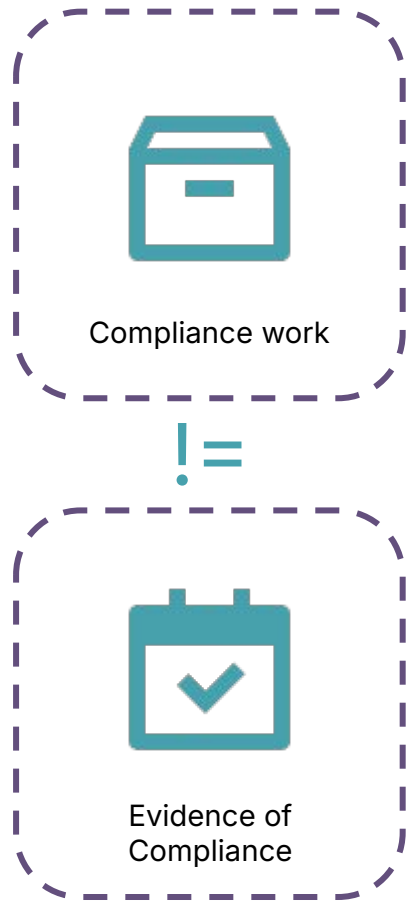
Tightly coupled db, caches, queues, etc managed by another team

Can you actually prove compliance?

Most organizations have no shortage of compliance policies, especially in highly regulated environments.

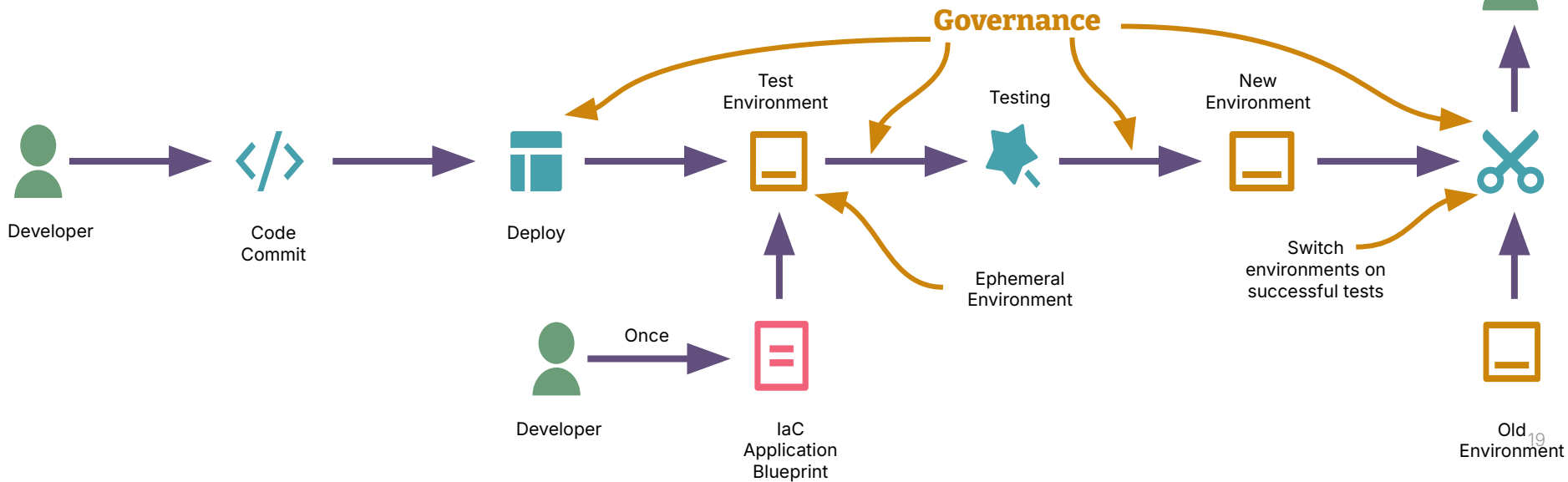
How do you actually ensure compliance? It's usually yearly training courses along with a bunch of architectural decision records. These may be "approved" by manual stage gates in a pipeline or architectural review board.

Rarely is there actual evidence of compliance at a meaningful level. For example. validation of Configuration Requested compared to the Configuration Performed.



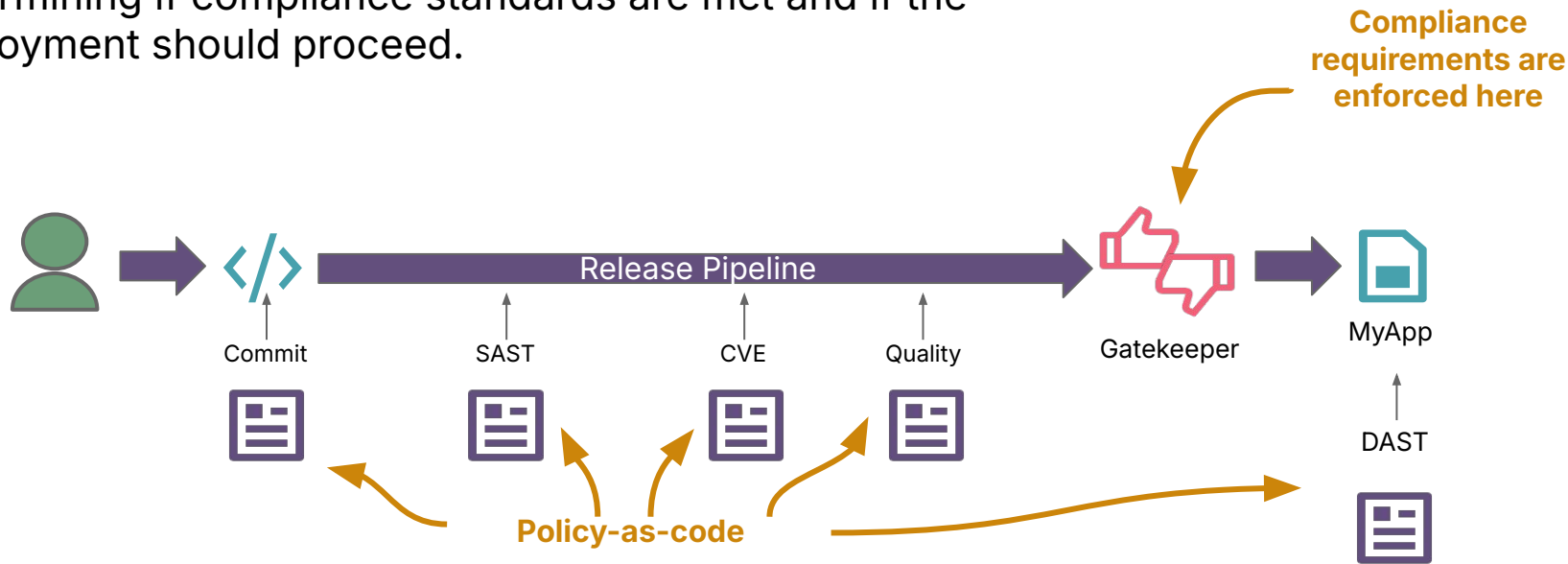
Governance at the point of change

Governance at the point of change is focused around leverage a policy-as-code engine that can provide governance across many areas of the software development lifecycle. This takes written policy and makes it enforceable in the real world.

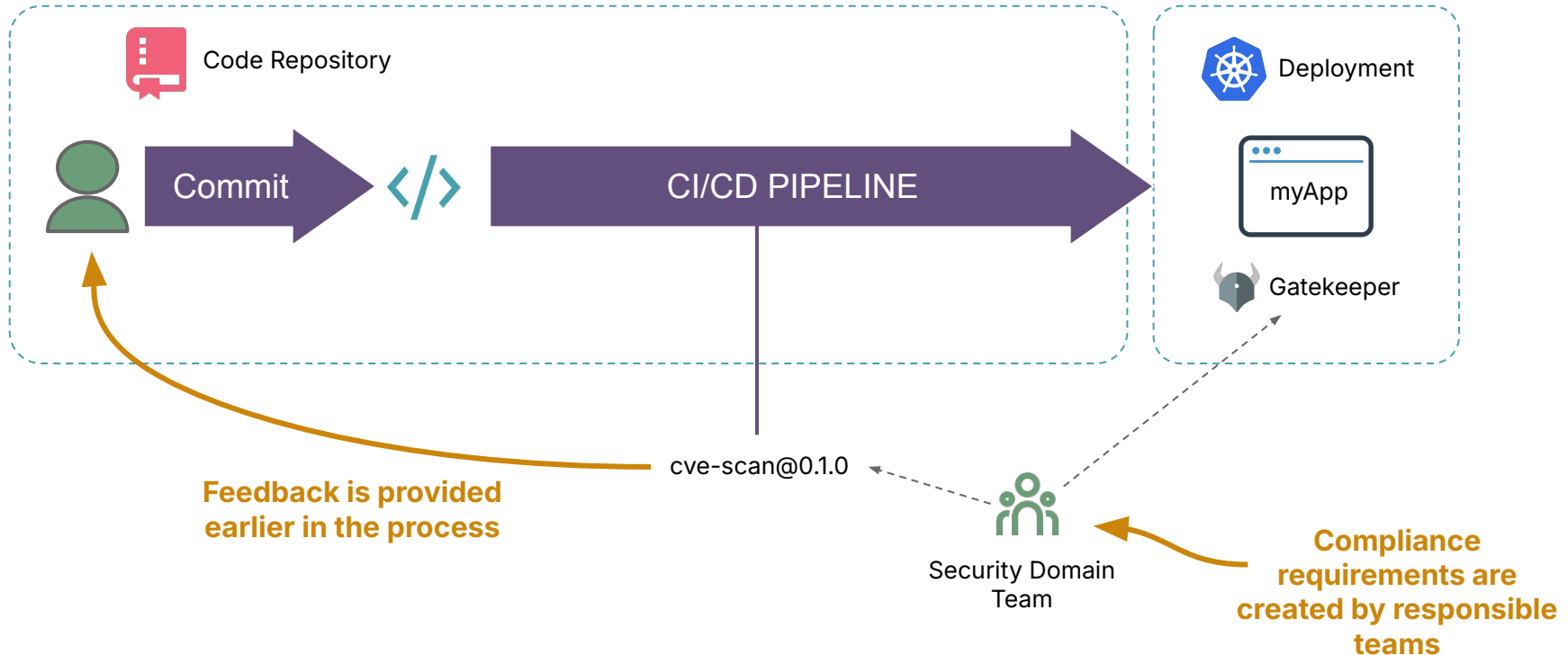


Policy-as-code: Gatekeepers

Gatekeepers are agents that sit in between development teams and the a given environment. They are responsible for taking a set of inputs and determining if compliance standards are met and if the deployment should proceed.



Centralized policies, federated enforcement

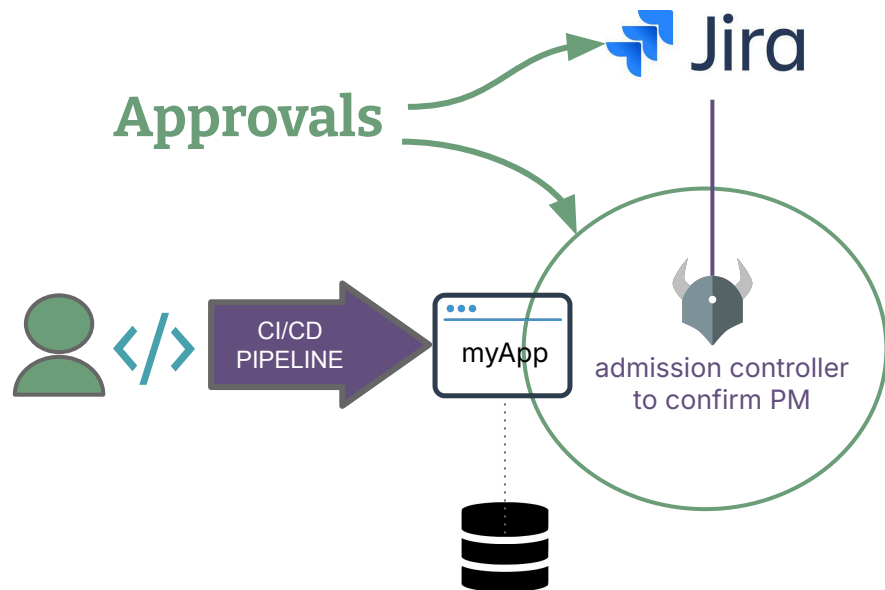


Point-of-Change Compliance: Approvals

Not all approval can be automated or approved via governance checks, especially in highly regulated areas.

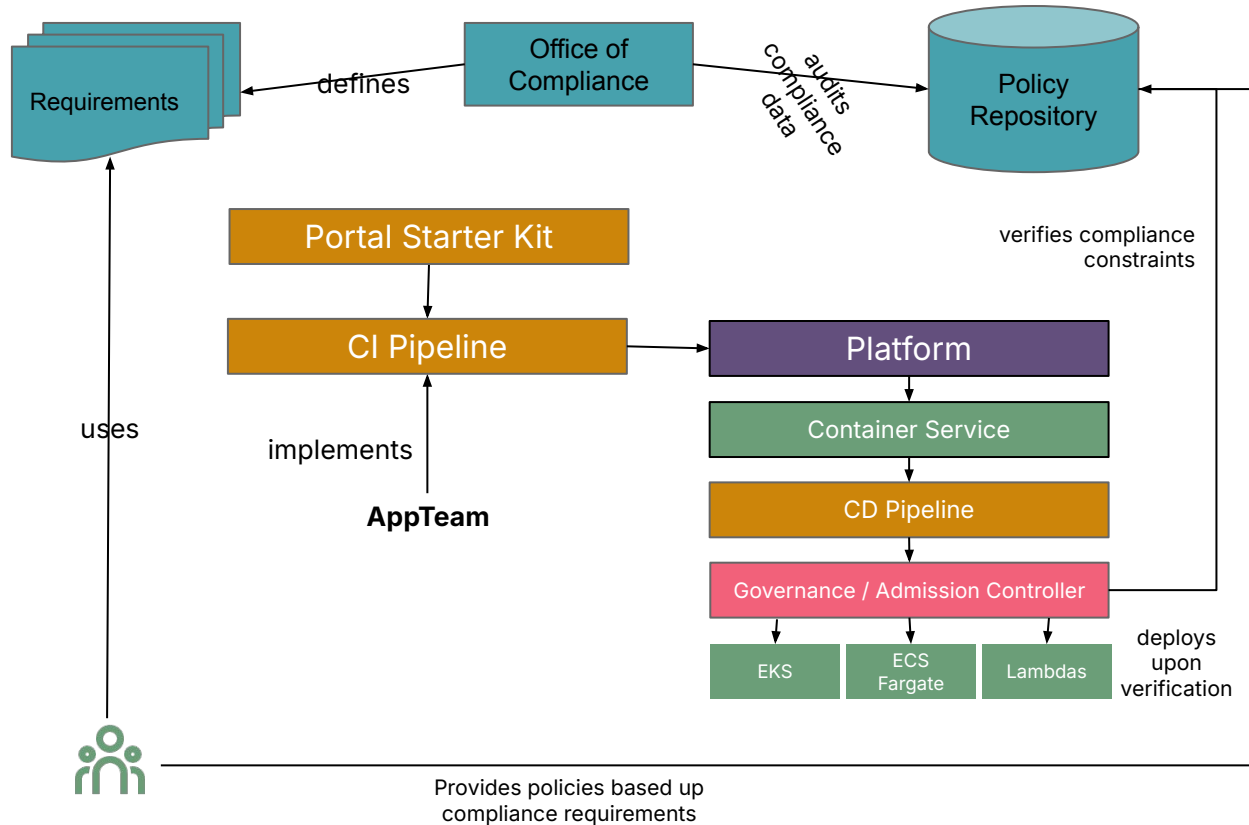
You can use the gatekeeper to validate approvals from tools like Jira or any tool that has an API.

The policies and gatekeeper can also be used for drive exception processes for critical updates like with outages.



Success Factors

Platforms streamline governance



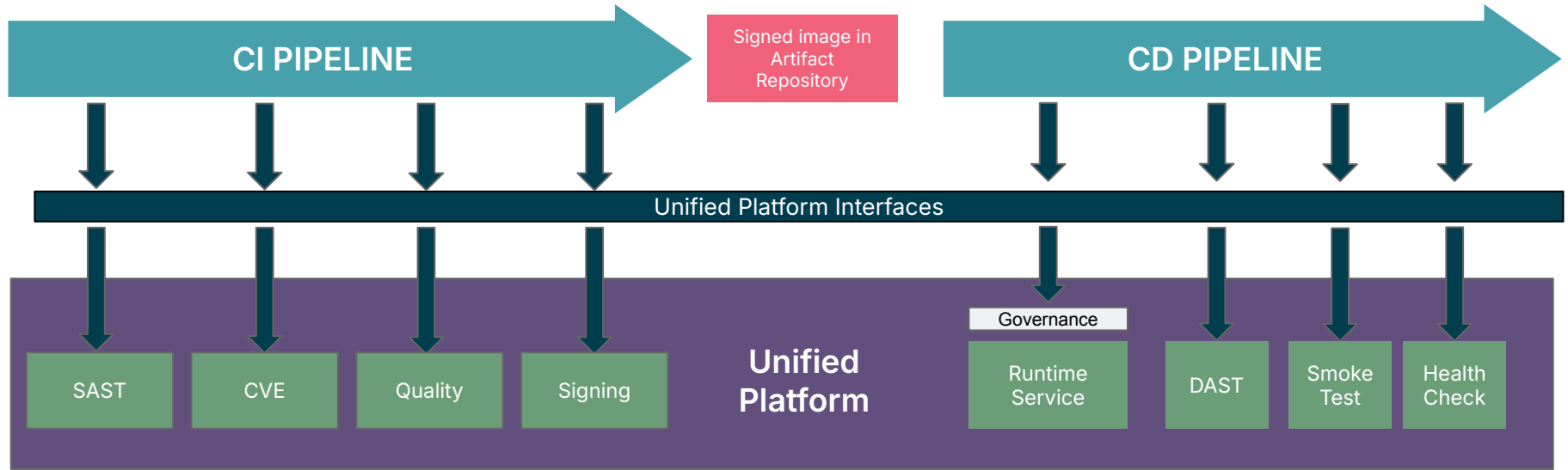
Confirmation of Evidence
from Assessment Before
Deployment

Validation of Configuration
Requested from
Configuration Performed

Deployment Teams Own
Pipeline

Compliance decoupled from
Deployment Teams

Platform interface simplifies software delivery



Standardized tooling with a common interface allows for an easy interface for application team and allows for platform teams to make updates in the background

Enabling Governance & Compliance

Guardrails should be used to ensure compliance to protect the company as well as ensuring teams are not subject to audit risk. The paved road and runtime environment include different checks and balances to ensure that lightweight governance can be applied in a fully automated manner

Static and dynamic code analysis	Code provenance	Security, governance, and Policy compliance	Links to agile workflow systems and changelogs
Code coverage, license verification, code quality, etc all show that code has been rigorously tested to minimize risk	Commits and container images are cryptographically signed to prove all changes are authorised and have passed all pipeline checks	Provides assurance that all applicable policy is followed to ensure low security and audit risk of changes	By having links to user stories for every change and automatically generating changelogs for every release, separation of concerns can be shown for SOX-compliant orgs

Guardrails enforced at the point of change



Policy as code engine

A policy as code engine (OPA) is used as an admission controller. This same engine is available to teams for other uses of OPA such as API authentication

Safeguards are applied to enforce required practices

The engine evaluates compliance to policies before code is allowed to deploy, ensuring conformance to guardrails of security, governance, and policy

Teams should still test for security and compliance

Teams should still test for these concerns as part of the pipeline to enable fast feedback loops. These controls should be considered the final gate

Governance at the "point of change"

Relevant Hypotheses

The number and complexity of ticket requests resulting in slow downs in software delivery

Low adoption of quality tools resulting in a higher number of defects or lower quality of code

Low adoption of security standards / tooling resulting in higher number of vulnerabilities

The lack of flexibility around how to evidence controls/compliance resulting in delays in production deployments

Lack of understanding of what controls are required resulting in delays in production deployments

Provisioned pipeline aren't able to be fully owned resulting in high cycle time for pipeline collaboration



Business Value

Decrease in time to market

Reduction in security risks

Improvement in software quality

Metrics to track

Time to Hello World in Prod

Time spent on compliance issue resolution

Day to onboard

NPS of standards, knowledge management

NPS of capabilities / products

Team Autonomy Factor (dependencies)

Cycle time of collaboration

% breakdown of existing high/medium/low security vulnerabilities

Rate of standard compliance issues

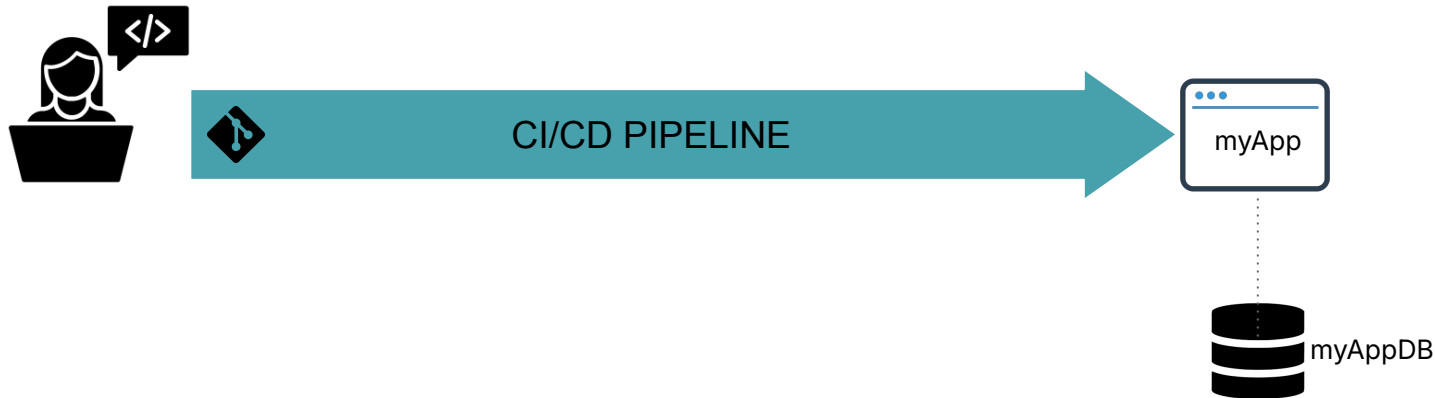
Adoption rate of platform services (security / quality tools)

Wait time on blocked by environment provisioning / change (production)

The Pattern in Detail

Maintaining Developer Ownership of the Pipeline

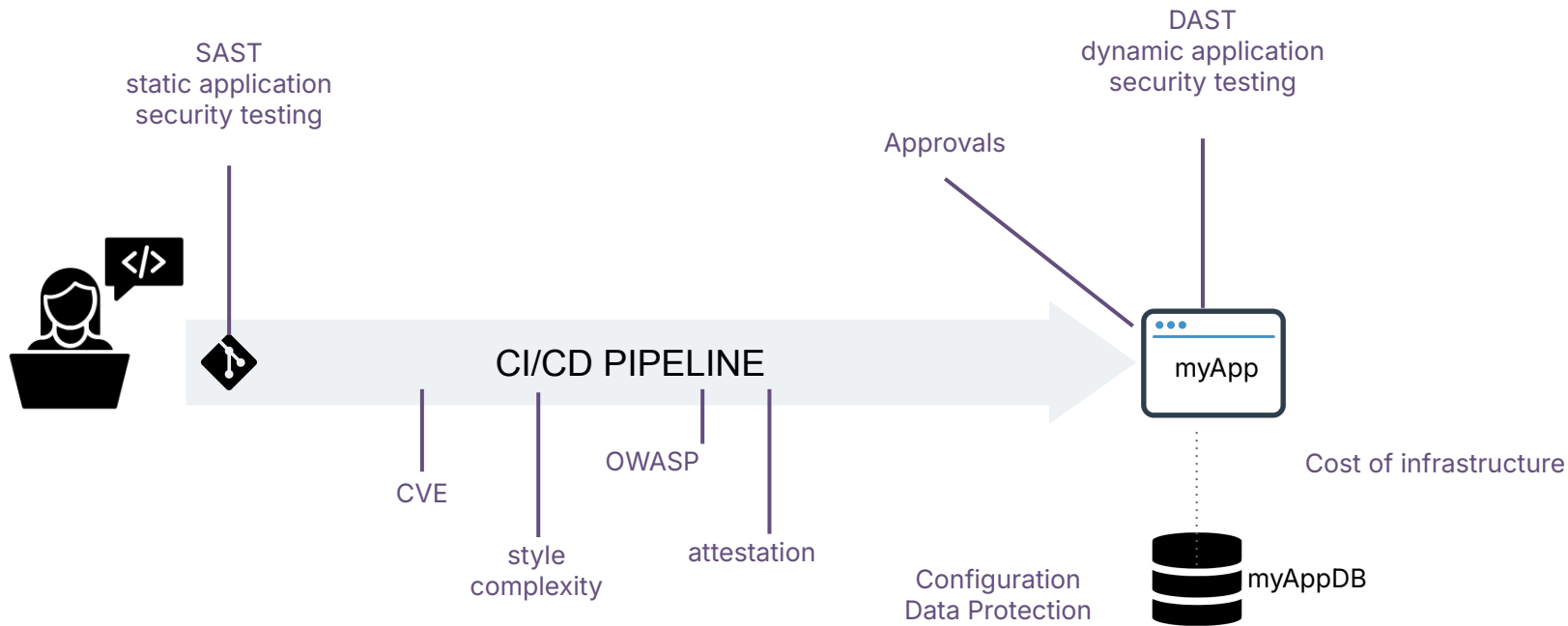
One of the MOST friction reducing architectural patterns available



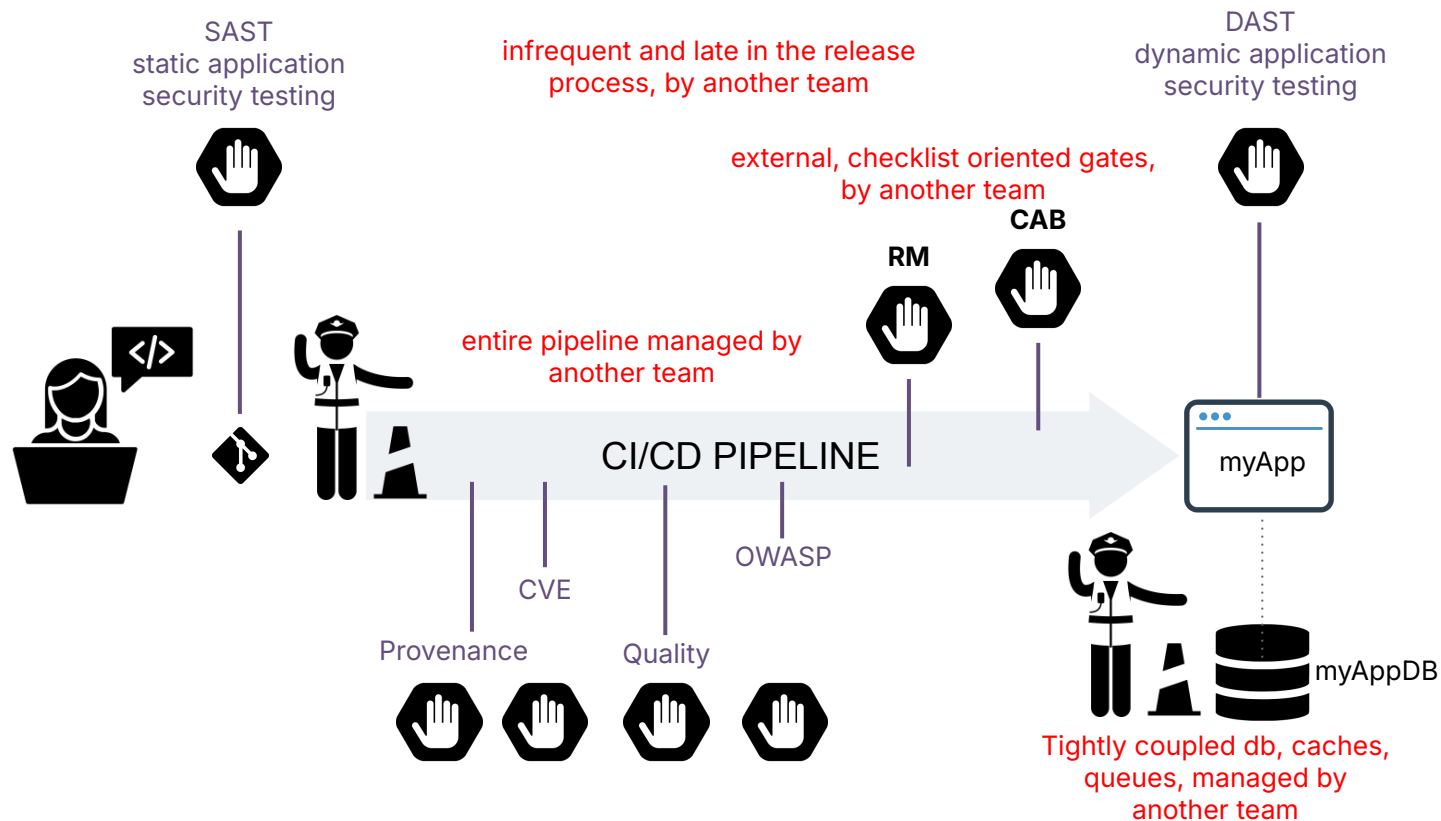
The *Everyday* Context



The *Everyday* Context

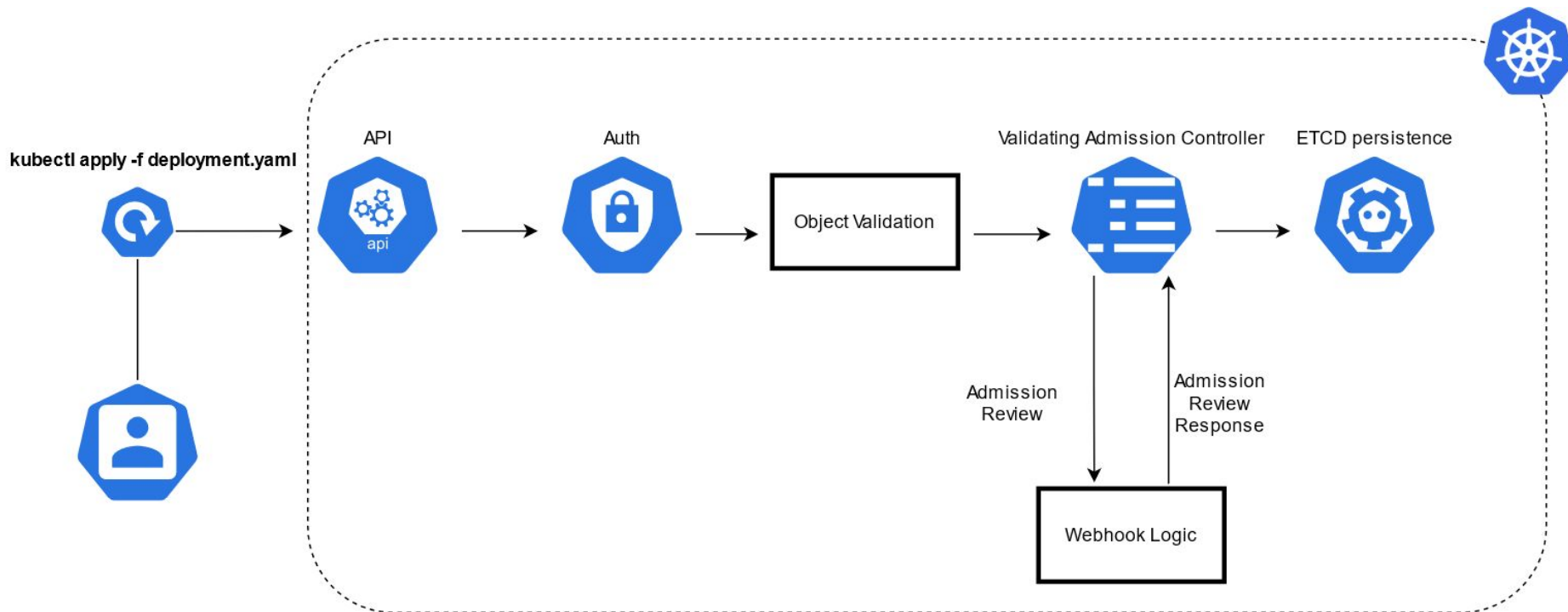


Compliance: **The norm**

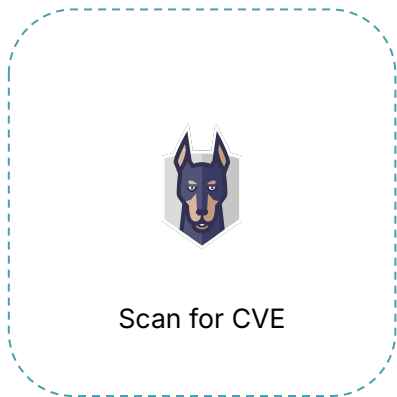


Rigorous verification of compliance

An example, the K8s Admission Controller



Ex: Remediate Known Package Vulnerabilities

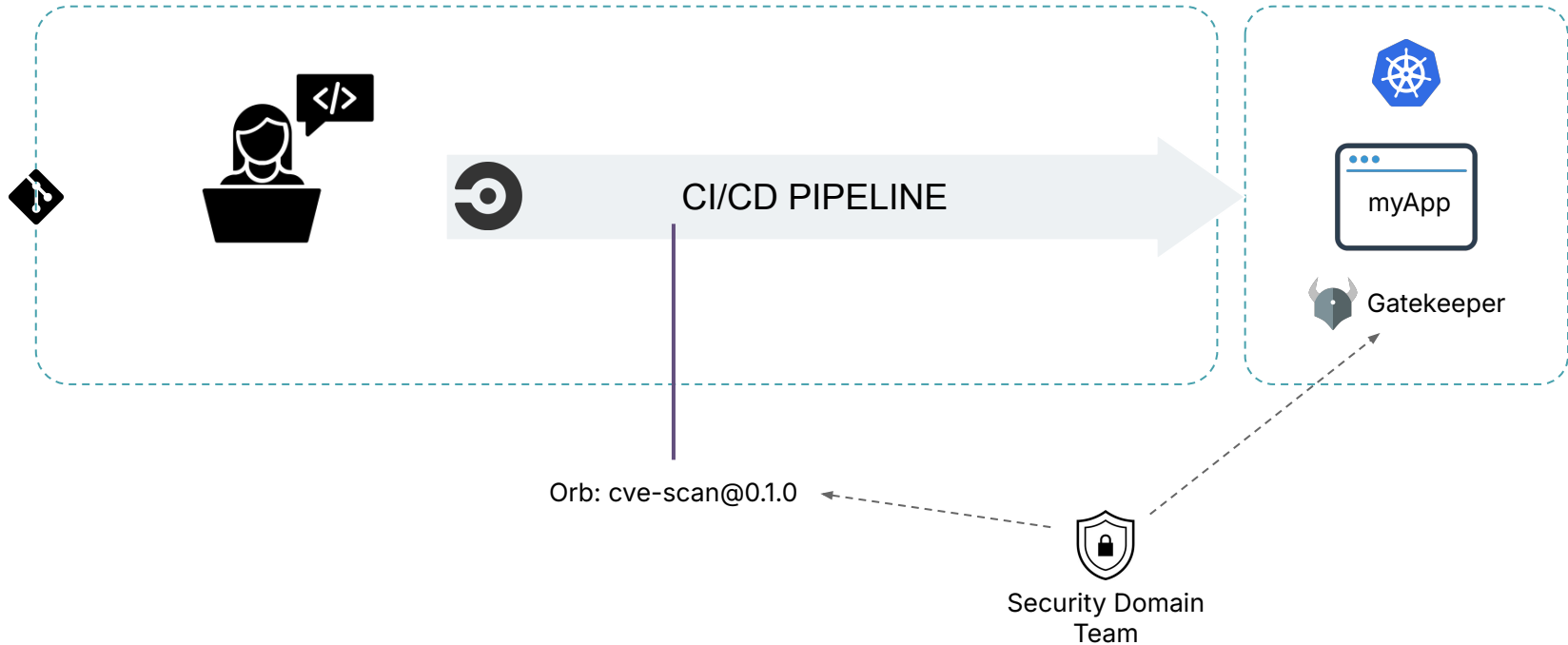


```
snyk test --docker myOrg/myImage:SHA2345234 \  
  --exclude-base-image-vulns \  
  --severity-threshold=low
```

...remediate

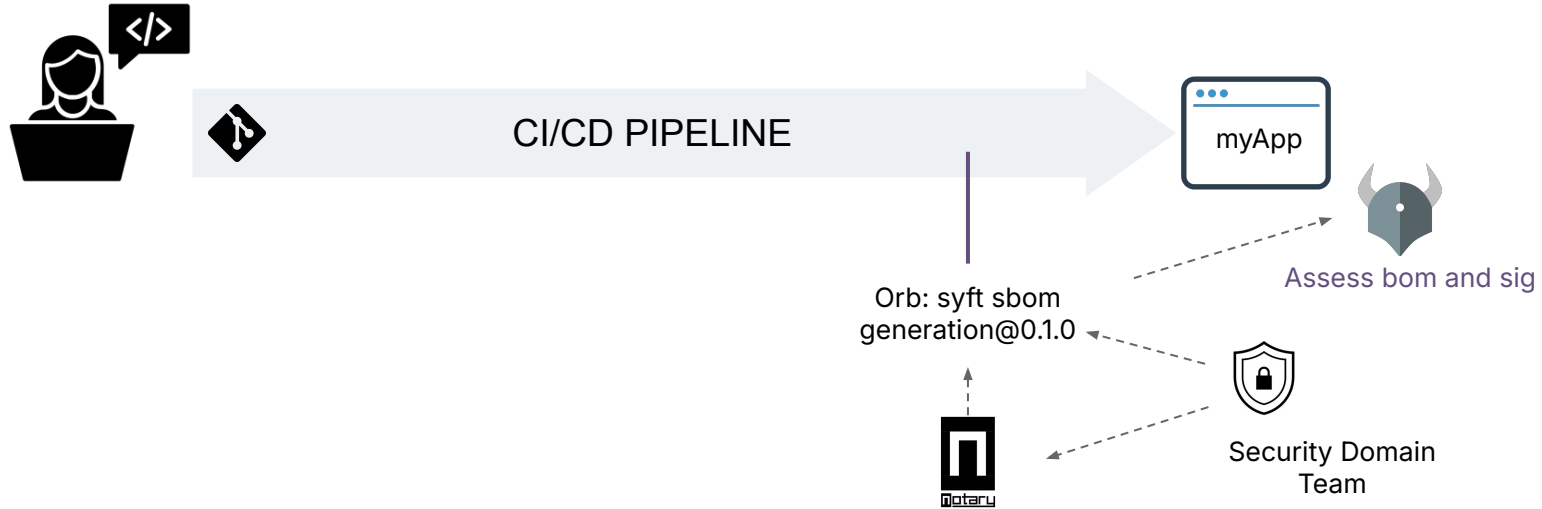
```
violation[{"msg": msg}] {  
  container := input_containers[_]  
  is_cve(container) # calls remote function to analyze snyk logs  
  msg := sprintf("container <%v> contains unaccepted CVE  
<%v>",  
    [container.name, container.image])  
}
```

Point-of-Change Compliance: Code Analysis

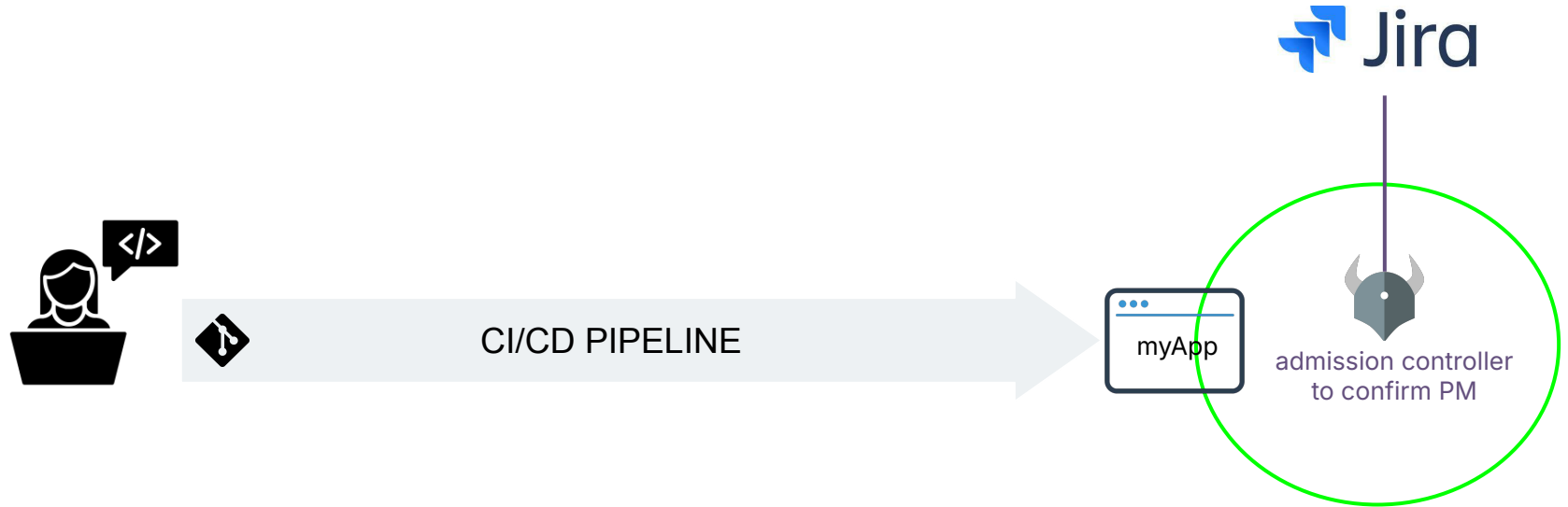


Point-of-Change Compliance: **Bill of Materials**

Artifact: OCI images



Point-of-Change Compliance: Approvals



Compliance at the Point of Change

Separate the work of compliant delivery
from the verification of being compliant



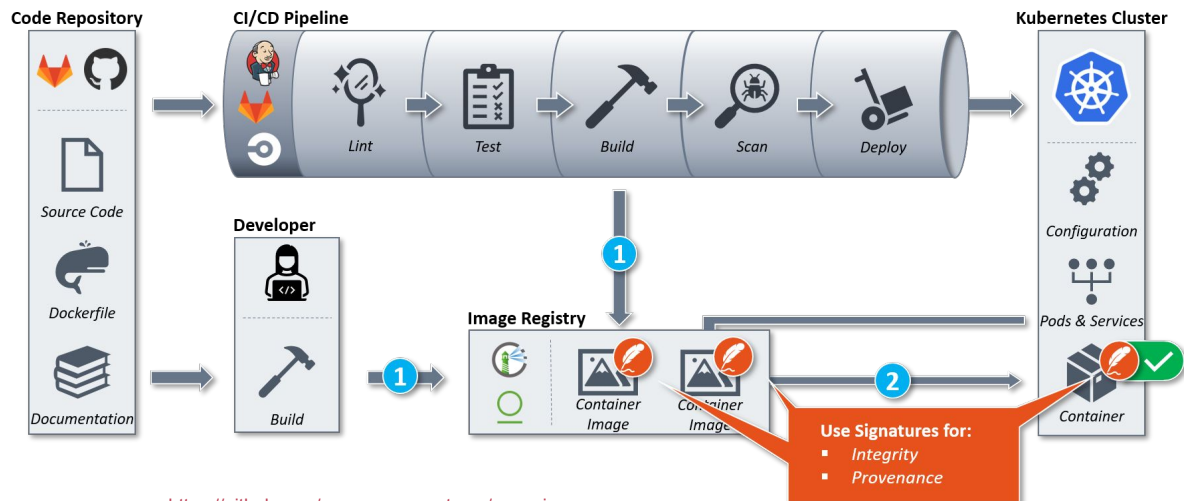
Other Considerations

Other considerations - Code Provenance

Code provenance is the ability to attest to the origin of any code running in the system. A strategy to accomplish this is to ensure that any images deployed to the cluster are signed by a *cryptographically verifiable key*.

Signing should be performed using a known key only after a CI/CD pipeline completes the build with any needed testing performed.

This allows an admission controller to validate that any pod is defined using a trusted image, helping to mitigate injections of unknown origin.



Open Policy Agent / Gatekeeper

Here we will use OPA Gatekeeper as our admission controller

- Open Policy Agent (OPA) uses the REGO language to define policy
- OPA is our policy engine
- OPA can be enabled across the technology landscape
- Gatekeeper is a runtime that serves as our policy decision point based on the OPA policy evaluation
- Policies can include identity information
- This will ensure that any policy enforcement is done at the *point of change*



Policy Implementation

First we deploy a policy template to the cluster or agent that serves as the definition for policies we can create.

Some example policies could be

- Label defined
- Configurations defined (i.e. requests, limits, etc)
- Containers run as non-root
- Images from trusted repository with signed cryptography key

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
        listKind: K8sRequiredLabelsList
        plural: k8srequiredlabels
        singular: k8srequiredlabels
      validation:
        # Schema for the `parameters` field
        openAPIV3Schema:
          properties:
            labels:
              type: array
              items: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8srequiredlabels

        deny[{"msg": msg, "details": {"missing_labels": missing}}] {
          provided := {label | input.review.object.metadata.labels[label]}
          required := {label | label := input.parameters.labels[_]}
          missing := required - provided
          count(missing) > 0
          msg := sprintf("you must provide labels: %v", [missing])
        }
```

Policy: Remediate Known Package Vulnerabilities



Scan for CVE

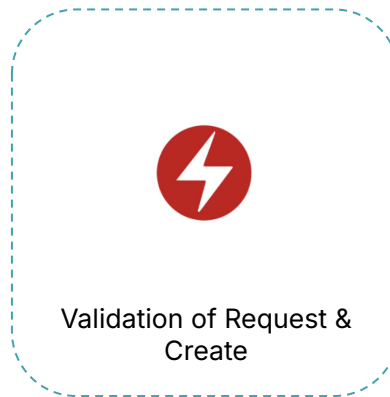


Block Deployment of app
with unaccepted CVE

```
snyk test --docker myOrg/myImage:SHA2345234 \  
--exclude-base-image-vulns \  
--severity-threshold=low
```

```
violation[{"msg": msg}] {  
    container := input_containers[_]  
    is_cve(container) # calls remote function to analyze snyk logs  
    msg := sprintf("container <%v> contains unaccepted CVE  
<%v>",  
        [container.name, container.image])  
}
```

Policy: Configuration Requirements for MySQL



```
apiVersion: rds.services.k8s.aws/v1alpha1
kind: DBCluster
metadata:
  name: $DB_CLUSTER_ID
spec:
  engine: aurora-mysql
```

```
package AuroraResource

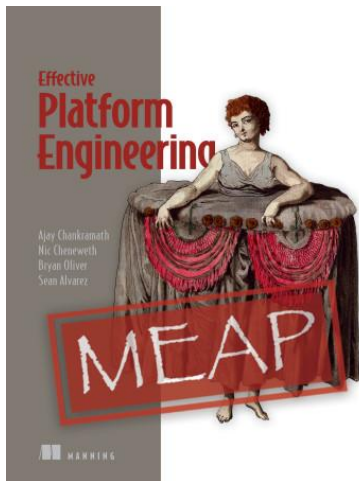
const (
    engineMode String =
    serverless
)
```

Further Reading:

<https://martinfowler.com/articles/devops-compliance.html>

Effective Platform Engineering


- **Effective Platform Engineering** book by [Manning](#)
- MEAP currently out awaiting print version soon





More Information



<https://effectiveplatformengineering.com/>



Effective Platform Engineering





[Buy Now](#)


[Ajay Chankramath](#)



[Nic Cheneweth](#)



[Bryan Oliver](#)



[Sean Alvarez](#)


"Effective Platform Engineering" is a comprehensive guide that introduces platform engineering as a discipline, focusing on creating developer platforms that enhance team efficiency and streamline application deployment. The book provides practical insights into designing and managing platforms that bridge the gap between operations and development, automating tasks throughout the software development lifecycle. Readers will learn to build internal developer platforms and portals, ensuring seamless adoption and satisfaction among teams. The authors emphasize the importance of secure, scalable Kubernetes-based engineering platforms and offer strategies for implementing effective Service Level Objectives to boost trust and adoption. Additionally, the book explores cutting-edge integrations of Generative AI tools to enhance developer productivity, providing readers with the knowledge to leverage the latest advancements in code generation.

Through practical examples and real-world scenarios, "Effective Platform Engineering" demonstrates how platform engineering differs from traditional DevOps and the unique value it brings to organizations. The book delves into both patterns and anti-patterns of platform development, guiding readers in designing and deploying secure, scalable, and observable engineering platforms. With the inclusion of diagrams, code samples, and exercises, readers can visualize key concepts and solidify their understanding. This resource is tailored for DevOps engineers familiar with Kubernetes, cloud environments, and infrastructure-as-code, aiming to equip them with the skills to establish platforms that reduce workloads, improve consistency, and accelerate software delivery.

Discover how platform engineering is revolutionizing the developer experience and operational efficiency. [Learn more](#)